CSE 2320 Notes 8: Sorting

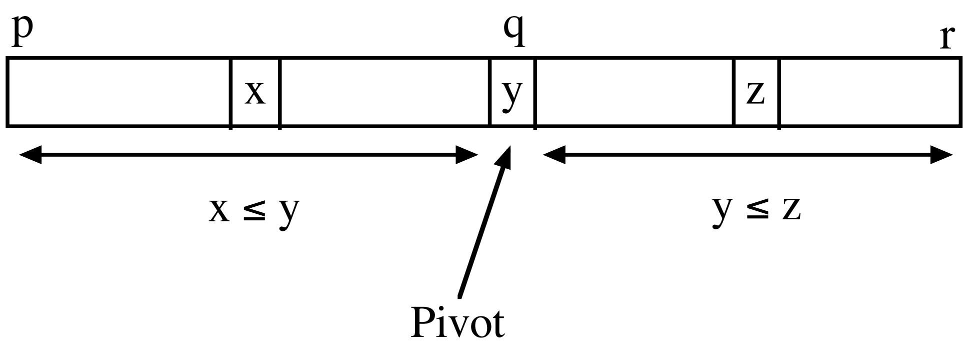
(Last updated 10/3/18 7:16 PM)

CLRS 7.1-7.2, 9.2, 8.1-8.3

8.A. QUICKSORT

Concepts

Idea: Take an unsorted (sub)array and *partition* into two subarrays such that



Customarily, the last subarray element (subscript r) is used as the *pivot* value.

After partitioning, each of the two subarrays, p . . . q – 1 and q + 1 . . . r, are sorted recursively.

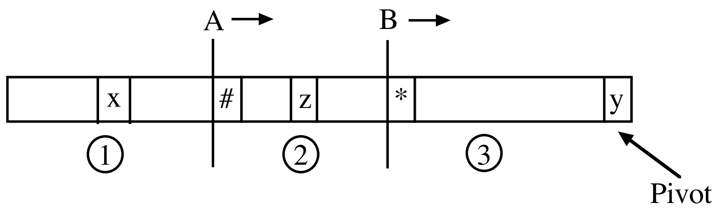
Subscript q is returned from PARTITION (aside: some versions don't place pivot in its final position).

Like MERGESORT, QUICKSORT is a divide-and-conquer technique:

	MERGESORT	QUICKSORT
Divide	Trivial	PARTITION (in-place)
Subproblems	Sort Two Parts	Sort Two Parts
Combine	MERGE (not in-place)	Trivial
Bottom-up possible?	Yes	No

http://ranger.uta.edu/~weems/NOTES2320/qsortRS.c

Version 1: PARTITION (in $\Theta(n)$ time, see `http://ranger.uta.edu/~weems/NOTES2320/partition.c`)



① Already known to have x ≤ y

② Already known to have y < z

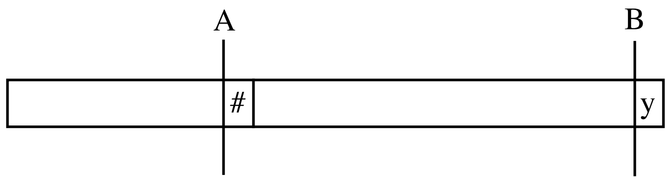
③ Untouched

y < *: Move B over

* ≤ y: Swap # & *
Move B over
Move A over

A and B can be at the the same position . . .

Termination



Swap # & y to place y in its final position.

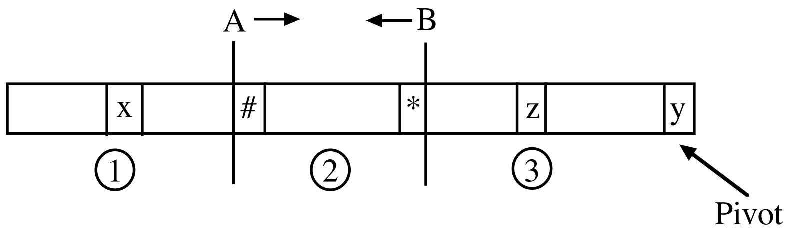
```
int newPartition(int arr[],int p,int r)
// From CLRS, 2nd ed.
{
int x,i,j,temp;

x=arr[r];
i=p-1;
for (j=p;j<r;j++)
  if (arr[j]<=x)
  {
    i++;
    temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;
  }
temp=arr[i+1];
arr[i+1]=arr[r];
arr[r]=temp;
return i+1;
}
```

Example:

```
AB 6    3    7    2    8    4    9    0    1    5
A  6  B 3    7    2    8    4    9    0    1    5
   3 A  6  B 7    2    8    4    9    0    1    5
   3 A  6    7  B 2    8    4    9    0    1    5
   3    2 A  7    6  B 8    4    9    0    1    5
   3    2 A  7    6    8  B 4    9    0    1    5
   3    2    4 A  6    8    7  B 9    0    1    5
   3    2    4 A  6    8    7    9  B 0    1    5
   3    2    4    0 A  8    7    9    6  B 1    5
   3    2    4    0    1 A  7    9    6    8  B 5
   3    2    4    0    1  < 5 >  9    6    8    7
```

Version 2 (Aside: Sedgewick, similar to CLRS, p. 185): Pointers move toward each other (also in $\Theta(n)$ time, see `http://ranger.uta.edu/~weems/NOTES2320/partitionRS.c`)



① Already known to have x ≤ y

③ Already known to have y ≤ z

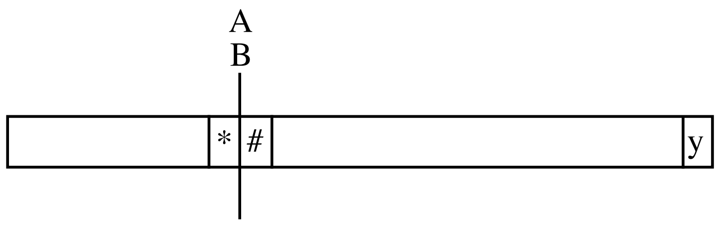
② Untouched

ⓐ # < y: Move A right

ⓑ y < *: Move B left

ⓒ Swap # and * (unless A and B have collided)

Termination



Swap # & y to place y in its final position.

```
int partition(Item *a,int ell,int r)
{
// From Sedgewick, but more complicated since pointers move
// towards each other.
// Elements before i are <= pivot.
// Elements after j are >= pivot.
int i = ell-1, j = r; Item v = a[r];

printf("Input\n");
dump(arr,ell,r);

for (;;)
  {
    // Since pivot is the right end, this while has a sentinel.
    // Stops at any element >= pivot
    while (less(a[++i], v)) ;
    // Stops at any element <= pivot (but not the pivot) or at the left end
    while (less(v, a[--j])) if (j == ell) break;
    if (i >= j) break; // Don't need to swap
    exch(a[i], a[j]);
  }
exch(a[i], a[r]);     // Place pivot at final position for sort
return i;
}
```

Examples:

```
A  6    3    7    2    8    4    9    0    1    5 B Left positioned
A  6    3    7    2    8    4    9    0    1  B 5   Right positioned
A  1    3    7    2    8    4    9    0    6  B 5   After swap
   1 A  3    7    2    8    4    9    0    6  B 5   Left continues
   1    3 A  7    2    8    4    9    0    6  B 5   Left positioned
   1    3 A  7    2    8    4    9    0  B 6    5   Right positioned
   1    3 A  0    2    8    4    9    7  B 6    5   After swap
   1    3    0 A  2    8    4    9    7  B 6    5   Left continues
   1    3    0    2 A  8    4    9    7  B 6    5   Left positioned
   1    3    0    2 A  8    4    9  B 7    6    5   Right continues
   1    3    0    2 A  8    4  B 9    7    6    5   Right positioned
   1    3    0    2 A  4    8  B 9    7    6    5   After swap
   1    3    0    2    4 A  8  B 9    7    6    5   Left positioned
   1    3    0    2    4 AB 8    9    7    6    5   Pointers collided
   1    3    0    2    4  < 5>   9    7    6    8   Pivot positioned
```

```
A  9    8    7    6    5    1    2    3    4    5 B Left positioned
A  9    8    7    6    5    1    2    3    4  B 5  Right positioned
A  4    8    7    6    5    1    2    3    9  B 5  After swap
   4 A  8    7    6    5    1    2    3    9  B 5  Left positioned
   4 A  8    7    6    5    1    2    3  B 9    5  Right positioned
   4 A  3    7    6    5    1    2    8  B 9    5  After swap
   4    3 A  7    6    5    1    2    8  B 9    5  Left positioned
   4    3 A  7    6    5    1    2  B 8    9    5  Right positioned
   4    3 A  2    6    5    1    7  B 8    9    5  After swap
   4    3    2 A  6    5    1    7  B 8    9    5  Left positioned
   4    3    2 A  6    5    1  B 7    8    9    5  Right positioned
   4    3    2 A  1    5    6  B 7    8    9    5  After swap
   4    3    2    1 A  5    6  B 7    8    9    5  Left positioned
   4    3    2    1 A  5  B 6    7    8    9    5  Pointers collided
   4    3    2    1  < 5>   6    7    8    9    5  Pivot positioned
```

QUICKSORT Analysis [Aside: also applies to the binary search trees of Notes 11]

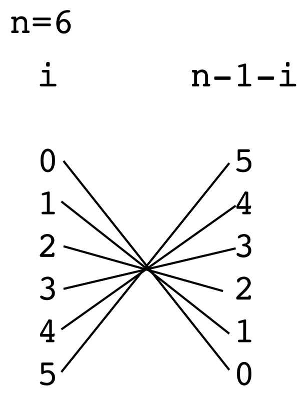
Worst Case – Pivot is smallest or largest key in subarray *every time*. (Includes ascending or descending order.) Let $T(n)$ be the number of comparisons.

$$T(n) = T(n-1) + n - 1 = \sum_{i=1}^{n-1} i = \Theta\left(n^2\right)$$

Best Case – Pivot ("median") always ends up in the middle. $T(n) = 2T\left(\frac{n}{2}\right) + n - 1$ (Similar to mergesort.)

Expected Case – Assume all $n!$ permutations are equally likely to occur. Likewise, each element is equally likely to occur as the pivot (each of the n elements will be the pivot in $(n-1)!$ cases). $E(n)$ is the expected number of comparisons. $E(0) = 0$.

$$E(n) = n - 1 + \sum_{i=0}^{n-1} \frac{1}{n}\left(E(i) + E(n-1-i)\right) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} E(i)$$



Show $\mathrm{O}(n\log n)$. Suppose $E(i) \le ci\ln i$ for $i < n$.

$$E(n) \le n - 1 + \frac{2c}{n}\sum_{i=1}^{n-1} i\ln i \le n\text{-}1 + \frac{2c}{n}\int_1^n x\ln x\,dx$$ [Bound above by integral]

$$= n - 1 + \frac{2c}{n}\left[\frac{1}{2}x^2\ln x - \frac{x^2}{4}\right]_1^n$$ [From http://integrals.wolfram.com]

$$= n - 1 + \frac{2c}{n}\left(\frac{n^2}{2}\ln n - \frac{n^2}{4} + \frac{1}{4}\right) = n - 1 + cn\ln n - \frac{cn}{2} + \frac{c}{2n}$$

$$\le cn\ln n \text{ for } c \ge 2$$

Other issues:

Unbalanced partitioning also leads to worst-case *stack depth* in $\Theta(n)$.

Small subfiles - use simpler sort on each subfile or delay until quicksort finishes.

Pivot selection - random, median-of-three

Subfile with all keys equal for version 1 and 2 partitioning?

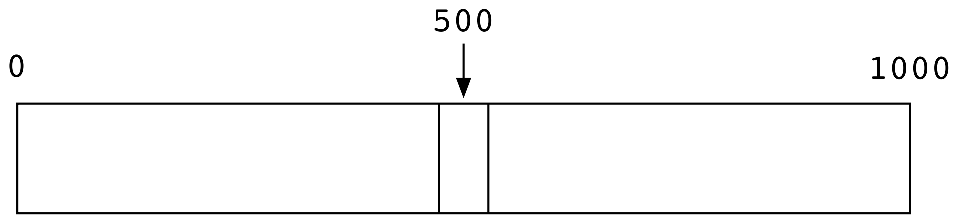
8.B. SELECTION AND RANKING USING QUICKSORT PARTITIONING IDEAS

Finding *k*th largest (or smallest) element in unordered table of *n* elements

(Aside: If *k* is small, e.g. $O\left(\frac{n}{\log n}\right)$, use a heap.)

Sort everything?

Use PARTITION several times. Always throw away the subarray that cannot include the target.



`http://ranger.uta.edu/~weems/NOTES2320/selection.c (quickSelection)`

`http://ranger.uta.edu/~weems/NOTES2320/klargest.c (quickLargest)`

$\Theta\left(n^2\right)$ worst case (e.g. input ordered)

$\Theta(n)$ expected. Let $E(k,n)$ represent the expected number of comparisons to find the *k*th largest in a set of *n* numbers. (Assume all $n!$ permutations are equally likely.)

Suppose $n = 7$ and $k = 3$. After 6 comparisons to place a pivot, the 7 possible pivot positions require different numbers of additional comparisons:

1	$E(3,6)$
2	$E(3,5)$
3	$E(3,4)$
4	$E(3,3)$
5	0
6	$E(1,5)$
7	$E(2,6)$

Suppose $n = 8$ and $k = 6$. After 7 comparisons to place a pivot, the 8 possible pivot positions require different numbers of additional comparisons:

1	$E(6,7)$
2	$E(6,6)$
3	0
4	$E(1,3)$
5	$E(2,4)$
6	$E(3,5)$
7	$E(4,6)$
8	$E(5,7)$

Observation: Finding the median is slightly more difficult than all other cases.

$$E(k,n) = n-1+\frac{1}{n}\sum_{i=1}^{k-1} E(i,n-k+i)+\frac{1}{n}\sum_{i=k}^{n-1} E(k,i)$$

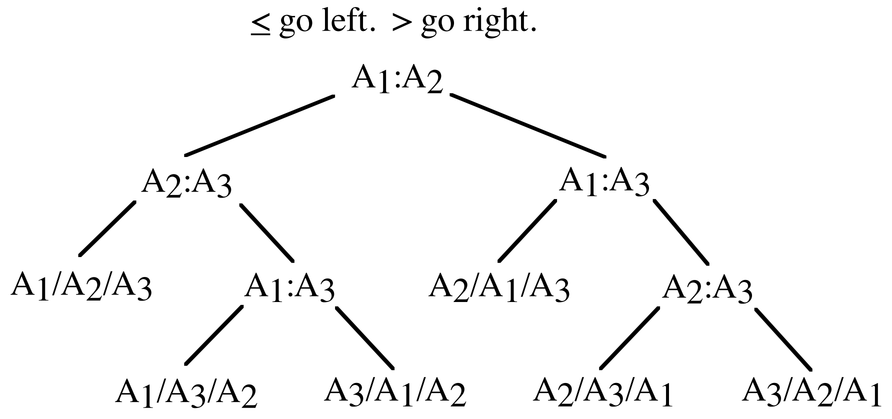
Show O(n). Using substitution method, suppose $E(i,j) \le cj$ for $j < n$.

$$\begin{aligned}
E(k,n) &\le n-1+\frac{1}{n}\sum_{i=1}^{k-1} c(n-k+i)+\frac{1}{n}\sum_{i=k}^{n-1} ci \\
&= n-1+\frac{c}{n}\sum_{i=1}^{k-1}(n-k+i)+\frac{c}{n}\sum_{i=k}^{n-1} i \\
&= n-1+\frac{c}{n}\sum_{i=1}^{k-1}(n-k)+\frac{c}{n}\sum_{i=1}^{k-1} i+\frac{c}{n}\sum_{i=k}^{n-1} i \\
&= n-1+\frac{c}{n}(k-1)(n-k)+\frac{c}{n}\sum_{i=1}^{n-1} i = n-1+\frac{c}{n}(k-1)(n-k)+\frac{c}{n}\frac{(n-1)n}{2} \\
&\le n-1+\frac{c}{n}\left(\frac{n^2}{4}-\frac{n}{2}+\frac{1}{4}\right)+\frac{c}{2}(n-1) \qquad k=\frac{n+1}{2} \text{ maximizes } \frac{c}{n}(k-1)(n-k) \\
&= n-1+\frac{cn}{4}-\frac{c}{2}+\frac{c}{4n}+\frac{cn}{2}-\frac{c}{2} = n-1+\frac{3cn}{4}-c+\frac{c}{4n} = cn-\frac{cn}{4}+n-1+\frac{c}{4n}-c \\
&\le cn \text{ for } c \ge 4
\end{aligned}$$

8.C. Lower Bounds on Sorting

Since a lower bound on a *problem* is to apply to a number of algorithms, it is necessary to have a *model* that captures the essential features of those algorithms. It is possible for algorithms to exist that do not follow the model.

Example Decision Tree



Decision Tree Model for Sorting

Two keys may be compared in $\Theta(1)$ time.

The time for other processing is proportional to the number of comparisons.

All $n!$ possible input permutations must be successfully sorted. (Leaves are labeled to show how input array has been rearranged.)

A tree with outcomes as leaves and decisions as internal nodes may be constructed for an algorithm and a specific value of n.

Worst-case comparisons?

Expected comparisons?

What is the minimum height of a decision tree for sorting n keys?

Since there must be $n!$ leaves, then the height is $\Omega(\lg(n!)) = \Omega(n \lg n)$. [Notes 2.D]

Other Examples of Lower Bounds (aside)

Binary search on ordered table – n leaves for the outcomes. $\Omega(\lg n)$ lower bound.

(Searching unordered table? Use *adversary* instead.)

Problem: Give decision tree model for merging two ordered tables with n elements each.

1. Number of outcomes is based on:

   a. $2n$ elements in output table.
   b. n elements of output table will receive n elements of first table *in their original order* (but possibly separated by elements from second table).

2. Number of leaves = number of outcomes =

$$\binom{2n}{n} = \frac{(2n)!}{n!n!} = \frac{(2\bullet 4\bullet 6\bullet\cdots\bullet 2n)(1\bullet 3\bullet 5\bullet\cdots\bullet 2n-1)}{n!n!}$$

$$= \frac{2^n(1\bullet 2\bullet 3\bullet\cdots\bullet n)(1\bullet 3\bullet 5\bullet\cdots\bullet 2n-1)}{n!n!}$$

$$= \frac{2^n}{n}\prod_{i=1}^{n-1}\frac{2i+1}{i} \geq \frac{2^n}{n}\prod_{i=1}^{n-1}\frac{2i}{i} = \frac{2^{2n-1}}{n}$$

3. Height of tree is bounded below by lg(number of leaves) = $\lg\binom{2n}{n} \geq \lg\frac{2^{2n-1}}{n} = 2n-1-\lg n$.

8.D. Stable Sorting (review)

A sort is *stable* if two elements with equal keys maintain their original (input) order in the output.

Practical significance is for situations with a *compound key*:

1. Each time a user logs into a computer, a record is created with user name, date, and time.
2. Once a year, each user receives a chronological report listing their log-ins.
3. If a stable sort is available, then the sort for (2) can use just the user name as the sort key.

Which sorts can be coded “naturally” to achieve stability?

Selection Insertion Merge Heap Quick

How can an unstable sort be forced to behave like a stable sort?

8.E. Linear Time Sorting

If the range of keys is limited, then sorting by direct key comparisons might not be the fastest method.

Counting Sort – Sort n records with keys in range 0 . . . $k-1$.

1. Clear count table – one counter for each value in range. $\Theta(k)$

```
for (i=0; i<k; i++)
  count[i]=0;
```

2. Pass through input table – add to appropriate counter for each key. $\Theta(n)$

```
for (i=0; i<n; i++)
  count[inp[i]]++;
```

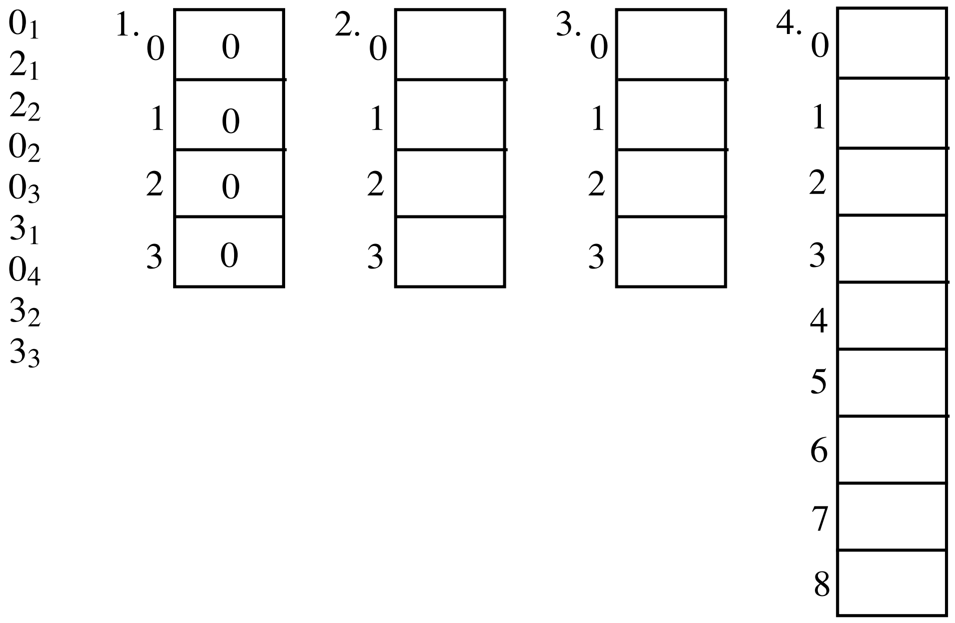
3. Determine first slot that will receive a record for each range value. $\Theta(k)$

```
slot[0]=0;
for (i=1; i<k; i++)
  slot[i]=slot[i-1]+count[i-1];
```

4. Copy each record to output, increment index in table from (3). $\Theta(n)$

```
for (i=0; i<n; i++)
  out[slot[inp[i]]++]=inp[i];
```

Overall, takes time $\Theta(k+n)$ which will be $\Theta(n)$ if k is bounded.



(LSD first) Radix Sort

Example: Sorting records whose keys are 9-digit Social Security Numbers.

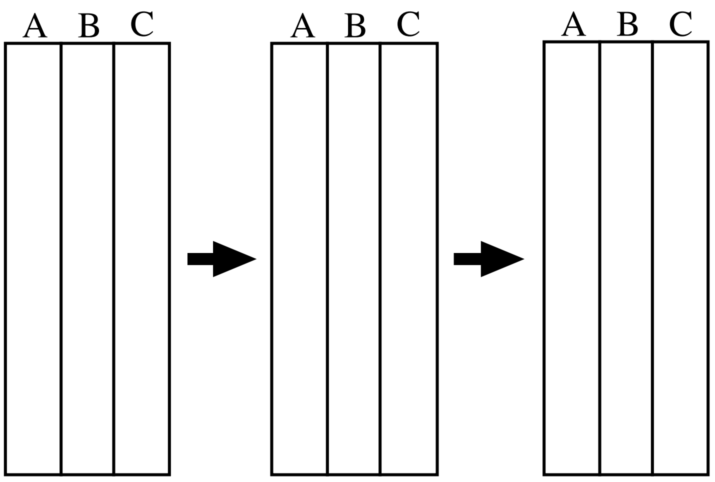
1. Treat each SSN as three digit number ABC where each digit is in the range 000 . . . 999.

```
c=ssn%1000;
b=(ssn/1000)%1000;
a=ssn/1000000;
```

2. Use counting sort to sort all records on C.

3. Use counting sort to sort all records on B. (Must be done in stable fashion.)

4. Use counting sort to sort all records on A. (Must be done in stable fashion.)



Time is $\Theta(d(k+n))$ where d is the number of digits (3), k is the size of the radix (1000), and n is the number of records.

Aside: In your favorite programming language, give general code for isolating a needed digit from a key.

Aside: d and k depend on each other

$$rangeSize = k^d \quad k = \sqrt[d]{rangeSize}$$

Inconvenient to compare asymptotically with key-comparison based sorts.

If the radix is binary, code similar to PARTITION may be used instead of counting sort.

Test Question: A billion numbers in the range 0 . . . 9,999,999 are to be sorted by LSD radix sort. How much faster will this be done if a decimal radix is used rather than a binary radix? Show your work.